

## TITLE OF THE INVENTION

Data Processor Having Translator and Interpreter That Execute  
Non-Native Instructions

## BACKGROUND OF THE INVENTION

### 5 Field of the Invention

The present invention relates to data processors, and particularly to a data processor including the function incorporated therein to execute non-native codes in addition to codes native to the processor.

### Description of the Background Art

10 Processors are designed and fabricated based on certain architectures. In the architecture, the instruction system, instruction format and the like are specified. The hardware is produced so as to execute instructions of such instruction formats efficiently. The program executed in a data processor is generally written in instruction codes of the  
15 mounted processor (referred to as "native code").

However, there is a case where a program written in an instruction code of a processor other than the mounted processor (referred to as "non-native code") is to be executed. For example, there may be the case where a program for an old processor with an old architecture is to be executed on a  
20 new processor with a new architecture when the program for that new processor is not yet fully developed. Another example is the case where a program written in the language such as Java™ bytecode ("Java" is a trademark of Sun Microsystems, Incorporation) for a virtual processor (called "Java virtual machine") is to be executed in various different types of  
25 apparatuses with different processors.

The conventional practice employed as the means for executing non-native code with a processor includes a software interpreter, a software translator and a hardware translator.

30 The procedure using a software interpreter is implemented by executing a series of processing steps set forth below with software called a software interpreter on the processor.

- (1) Read out non-native code from the memory.
- (2) Dispatch a relevant non-native code to the processing routine

associated with the readout non-native code.

(3) Execute the processing routine associated with the readout non-native code.

(4) Update the program counter of the non-native code.

5 This software interpreter per se is written in the code native to the mounted processor.

Software is so flexible that it can actually realize any process if the processing speed is not taken into account. Therefore, the software interpreter can be easily realized. However, there is a problem that the  
10 execution speed is degraded since the steps of (1), (2) and (4) are also required in addition to the actual process of step (3).

"Translator" implies a device that converts the program of non-native codes into an equivalent program of native codes. A hardware translator performs this conversion in hardware and a software translator  
15 performs this conversion with software.

A hardware translator is disclosed in, for example, U.S. Patent No. 5,875,336. According to the hardware translator, native codes to realize a process of equal contents to respective non-native codes are generated by hardware in order to simulate the operation of respective instructions of the  
20 non-native codes. However, the execution speed of the resultant native code of conversion is inevitably degraded by various reasons set forth in the following.

First, in order to read out the non-native code, the translator must simulate, not only the operation result, but also the PC (Program Counter)  
25 value of the non-native code or also the flags, if necessary. Therefore, the operation of one non-native code will be converted into a plurality of native codes.

Secondly, even if there are more registers provided in the processor than the expected number of registers provided in the processor based on the  
30 non-native code, the excessive registers cannot be utilized. For example, in the case of the translator, the memory operand of the non-native code is still converted into a memory operand in the native code, and is not allocated to the register.

One solution to the problem that the execution speed of the resultant native code after conversion is degraded is proposed in U.S. Patent No. 5,898,885. It is premised that the non-native code is a stack machine code such as Java bytecode. According to the proposed procedure, a native code  
5 is generated so that, when non-native code that pops data from a stack succeeds non-native code that pushes the data on the stack, these non-native codes will be executed together. Accordingly, the number of accesses to the memory is reduced. As a result, the execution speed can be improved. Similar art is proposed in U.S. Patent No. 6,026,485.

10 However, this invention has the disadvantage that the procedure cannot be applied unless there is a data popping step right after the data pushing step due to its constitutional limitation.

The above-described problem of speed degradation is also encountered in the conversion by a software translator. However, the  
15 software translator differs from the hardware translator in the flexibility of allowing the conversion process to be implemented in a larger program unit (subroutine unit or class unit) instead of the instruction unit. Since unnecessary memory access and the like can be eliminated in the software translator, speed degradation can be suppressed to a certain level.

20 However, the conversion process will become complicated if a native code that has little speed degradation during execution is to be generated and the conversion processing time will be increased.

In order to suppress such overhead of the conversion processing time, a possible consideration is the usage of the procedure to store in the memory  
25 the native codes once converted and generated by the software translator. When the same program portion is to be executed again, the conversion process is not carried out but the resultant native codes of conversion that are stored in the memory are used again. The conversion is skipped when the program portion that has been already converted is encountered at the  
30 second time and et seq. Therefore, the execution speed is increased compared to the case where conversion was carried out every time.

However, this procedure requires a great capacity of memory in order to store the resultant native codes after the program of the non-native

codes has been converted. As a result, another problem that the memory cost is increased will occur.

In order to suppress increase of the memory capacity for the storage of the resultant native codes of conversion, a possible consideration is to set the size of the RAM (Random Access Memory) that is used to store the resultant native codes of conversion constant. According to this procedure, the RAM is used likewise the so-called cache memory. In the present specification, this RAM is called "software cache".

According to this procedure, the non-native codes are converted into native codes in the subroutine (or method) unit to be additionally stored in the RAM. When this RAM becomes full, the native codes of a subroutine (or method) that has already been executed and has low possibility of being executed thereafter is dismissed from the RAM. The subroutine (or method) formed of new resultant native codes of conversion is stored in that released and available region of the RAM.

By the usage of such a software cache, it is expected that the software translator can have the increase of the memory capacity suppressed to a certain level. However, the number of conversion processes increases in comparison to the case where all the converted instructions are stored in the memory. As a result, the overhead will be increased. If conversion process is simplified to reduce the conversion processing time for example, if each instruction of non-native codes is converted one by one rather than several of them are converted as a group, the execution speed of the resultant native code after conversion will be degraded, as described before.

Thus, the software interpreter has the problem that the execution speed is significantly reduced. It is also difficult to prevent reduction in the execution speed for the hardware translator, though degradation of the execution speed is not as notable as for the software interpreter. There is also the problem that the hardware amount will be increased in order to translate all the non-native codes. Furthermore, it is noted that improving the execution speed and preventing memory increase are rather incompatible.

## SUMMARY OF THE INVENTION

In view of the foregoing, an object of the present invention is to provide a data processor including a translator of non-native codes that can execute non-native codes at high speed with a small amount of hardware.

5 According to an aspect of the present invention, a data processor includes a processor with a predetermined instruction group as a native code, a hardware translator converting a code which is non-native into one or more native codes of a processor, and a memory storing a program formed of a native code that operates on the processor. The program stored in the  
10 memory includes a software translator program operating on the processor to convert a code which is non-native to the processor into one or more native codes of the processor, a software interpreter program operating on the processor to sequentially interpret a code which is non-native to the processor and execute the interpreted code using a native code of the  
15 processor, and a selection program selecting the execution of a native code output by the hardware translator, or the execution of a native code output by the software translator and stored in the memory, or sequential interpretation and execution of non-native code by the execution of the software interpreter, according to a predetermined criterion to operate the  
20 processor.

Preferably, the selection program includes a program to select execution of a native code output by the hardware translator, or execution of a native code output by the software translator, or sequential interpretation and execution of non-native code by execution of the software interpreter,  
25 depending upon the type and execution frequency of the non-native code read out and the status of the memory to operate the processor.

Further preferably, the selection program includes a program selecting execution of a native code output by the hardware translator, or execution of a native code output by the software translator, or sequential  
30 interpretation and execution of non-native code by execution of the software interpreter, depending upon the type and execution frequency of the non-native code read out and the size of the available empty region in the memory to operate the processor.

Further preferably, the selection program includes a program to select execution of a native code output by the hardware translator, or execution of a native code output by the software translator, or sequential interpretation and execution of non-native code by execution of the software interpreter, depending upon the address in the memory and execution frequency of non-native code read out, and the size of the available and empty region in the memory to operate the processor.

The selection program may include a program selecting execution of a native code output by the hardware translator, or execution of a native code output by the software translator, or sequential interpretation and execution of non-native code by execution of the software interpreter, depending upon the address in the memory and execution frequency of non-native code read out to operate the processor.

The selection program may include a program selecting execution of a native code output by the hardware translator, or execution of a native code output by the software translator, or sequential interpretation and execution of non-native code by execution of the software interpreter every time a method constituted by non-native codes is called to operate the processor.

The software translator may include a code conversion program to convert non-native code into a native code so as to allocate at least a portion of a memory operand included in the non-native code to a register provided in the processor.

According to another aspect of the present invention, an operation method of a data processor including a processor with a predetermined instruction group as a native code, a hardware translator converting a code which is non-native to the processor into one or more native codes of the processor, and a memory storing a program formed of native codes operating on a processor is provided. The program stored in the memory includes a software translator program operating on the processor to convert a code which is non-native to the processor into one or more native codes of the processor and storing the converted code in the memory, and a software interpreter program operating on the processor to sequentially interpret a

code which is non-native to the processor and execute the interpreted code using the native code of the processor. The operation method includes the steps of selecting execution of a native code output by the hardware translator, or execution of a native code output by the software translator and stored in the memory, or sequential interpretation and execution of non-native code by execution of the software interpreter according to a predetermined criterion, and applying to the processor for operation a native code output by the hardware translator, or a native code output and stored in the memory by invoking the software translator with the readout non-native code as an argument, or a program code of the software interpreter with the readout non-native code as an argument, according to the selected result in the select step.

According to a further aspect of the present invention, a data processor includes a processor with a predetermined instruction group as a native code, a hardware translator converting non-native code to the processor into one or more native codes of the processor, a software translator operating on the processor to convert a code non-native to the processor into one or more codes native to the processor, a storage device storing a native code output from the software translator, a software interpreter operating on the processor to sequentially interpret codes non-native to the processor and executing the interpreted codes using native codes of the processor, and a select circuit selecting execution of a native code output by the hardware translator, or execution of a native code output by the software translator, or sequential interpretation and execution of non-native code by execution of the software interpreter, according to a predetermined criterion to operate the processor.

The foregoing and other objects, features, aspects and advantages of the present invention will become more apparent from the following detailed description of the present invention when taken in conjunction with the accompanying drawings.

#### BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 shows a data processor according to first-third embodiments of the present invention.

Figs. 2, 3 and 4 show an instruction set of a functional unit according to the first to third embodiments.

Fig. 5 is a block diagram of a multifunction instruction decoder 105.

5 Fig. 6 shows the method written in Java bytecode and the meaning of each Java bytecode. This method is used as the non-native code to be converted in the first to third embodiments.

Fig. 7 is a flow chart of the procedure which executes a program of non-native codes according to the first to third embodiments.

10 Fig. 8 is a flow chart of the procedure which invokes a subroutine of non-native codes according to the first to third embodiments.

Fig. 9 is a flow chart of the procedure which selects an execution mode according to the first to third embodiments.

Fig. 10 is a flow chart of the procedure of an execution process of non-native or native code according to the first to third embodiments.

15 Fig. 11 is a flow chart of the procedure which converts non-native code that requires complicated processing into a native code according to the first to third embodiments.

Fig. 12 is a flow chart of the procedure which converts invokestatic into a native code according to the first to third embodiments.

20 Fig. 13 is a flow chart of the procedure of a pre-execution preparation process according to the first to third embodiments.

Fig. 14 is a flow chart of the procedure which converts a subroutine of non-native codes into a subroutine of native codes according to the first to third embodiments.

25 Fig. 15 is a flow chart of the procedure of a conversion start process according to the first to third embodiments.

Fig. 16 is a flow chart of the procedure of a register allocation process according to the first to third embodiments.

30 Fig. 17 is a flow chart of the procedure of a registration process of the address and stack depth according to the first to third embodiments.

Fig. 18 is a flow chart of the procedure of a reference counting process with respect to an operand of jinst according to the first to third embodiments.

Fig. 19 is a flow chart of the procedure of a register allocation determination process according to the first embodiment.

Fig. 20 is a flow chart of the procedure of a method code conversion process according to the first to third embodiments.

5 Fig. 21 shows the native code generated for iconst\_<n> according to the first embodiment.

Fig. 22 shows the native code generated for iload\_<n> according to the first embodiment.

10 Fig. 23 shows the native code generated for istore\_<n> according to the first embodiment.

Fig. 24 shows the native code generated for iadd according to the first embodiment.

Fig. 25 shows the native code generated for ifge according to the first embodiment.

15 Fig. 26 shows the native codes generated for goto according to the first embodiment.

Fig. 27 shows the native codes generated for ireturn according to the first embodiment.

20 Fig. 28 shows the native codes generated for invokestatic according to the first embodiment.

Fig. 29 shows the usage of the M32R register according to the first embodiment.

Figs. 30-33 show the stack layout according to the first to third embodiments.

25 Fig. 34 shows the status of register allocation with respect to operands according to the first embodiment.

Figs. 35 and 36 show execution examples of the register allocation process according to the first embodiment.

30 Figs. 37 and 38 show execution examples of a method code conversion process according to the first embodiment.

Fig. 39 is a flow chart of the procedure of a register allocation determination process according to the second embodiment.

Fig. 40 is a flow chart of the procedure of a native code generation

process with respect to Java bytecode that does not branch according to the second embodiment.

Fig. 41 is a flow chart of the procedure of a native code generation process for ifge according to the second embodiment.

5        Fig. 42 is a flow chart of the procedure of a native code generation process for goto according to the second embodiment.

Fig. 43 shows the usage of the M32R register according to the second embodiment.

10       Fig. 44 is a flow chart of the procedure of a native code generation process accommodating overflow of registers  $S\langle p \rangle$ - $S\langle q \rangle$  according to the second embodiment.

Fig. 45 is a flow chart of the procedure of a native code generation process accommodating underflow of registers  $S\langle \text{bottom} \rangle$ - $S\langle \text{top} \rangle$  according to the second embodiment.

15       Fig. 46 shows the register allocation status with respect to operands according to the second embodiment.

Figs. 47 and 48 show execution examples of a method code conversion process according to the second embodiment.

20       Fig. 49 is a flow chart of the procedure of a native code generation process to store NS stack operands from  $S\langle j_{\text{next}}-1 \rangle$  in the register according to the second embodiment.

Fig. 50 is a flow chart of the procedure of a native code generation process for Java bytecode that effects only data transfer to a stack according to the third embodiment.

25       Fig. 51 is a flow chart of the procedure of a native code generation process to render  $P\langle i \rangle$  valid according to the third embodiment.

Fig. 52 is a flow chart of the procedure of a native code generation process to reserve register  $S\langle k \rangle$  according to the third embodiment.

30       Fig. 53 is a flow chart of the procedure of a native code generation process to store NS stack operands from  $S\langle k \rangle$  in a register according to the third embodiment.

Fig. 54 is a flow chart of the procedure of a native code generation process for iadd according to the third embodiment.

Fig. 55 shows the native codes generated for iadd according to the third embodiment.

Fig. 56 shows a native code generation process for junction at jpcnext according to the third embodiment.

5 Fig. 57 is a flow chart of the procedure of a native code generation process for invokestatic according to the third embodiment.

Fig. 58 is a flow chart of the procedure of a native code generation process for ireturn according to the third embodiment.

10 Fig. 59 is a flow chart of the procedure of a native code generation process for istore\_<n> according to the third embodiment.

Fig. 60 shows the purge process of L<n> according to the third embodiment.

Fig. 61 is a flow chart of the procedure of a native code generation process for goto according to the third embodiment.

15 Fig. 62 is a flow chart of the procedure of a native code generation process for ifge according to the third embodiment.

Figs. 63 and 64 show execution examples of a method code conversion process in the data processor of the third embodiment.

#### DESCRIPTION OF THE PREFERRED EMBODIMENTS

20 All the data processors according to respective embodiments of the present invention described hereinafter include a software translator converting Java bytecode into native codes by software, a hardware translator converting Java bytecode into native codes by software, a software interpreter analyzing and simulating Java bytecode in software  
25 during execution, and a selector automatically switching these three means depending upon the contents of the Java bytecode of interest. It is understood by a person skilled in the art that the present invention is not limited to these embodiments. For example, although Java bytecode is taken as the non-native code in respective embodiments, a translator can be  
30 realized in a similar manner in the case where a code other than the Java bytecode is executed as non-native code.

#### First Embodiment

Referring to Fig. 1, a data processor according to a first embodiment

of the present invention includes a processor 101, a RAM 102, and a ROM (Read Only Memory) 103 storing non-native codes at predetermined addresses, all connected to each other through a bus 107.

Processor 101 formed of a single semiconductor chip includes a processing unit 106 which is the main unit of processor 101 and has the function equal to that of a conventional processor, and a multifunction instruction decoder 105 sits between processing unit 106 and bus 107 to convert the instruction received from bus 107 into the native code of processing unit 106 when the received instruction is non-native code read out from a predetermined address of ROM 103 and provide the converted instruction to processing unit 106, or to directly provide the received instruction to processing unit 106 when the received instruction is not non-native code read out from a predetermined address of ROM 103.

RAM 102 serves to store the method of the native code of processing unit 106 and the data used by the method.

ROM 103 is a computer (processor 101) readable recording medium that stores a program which is executed by processor 101 and implements the procedure of the method call of non-native codes (including a program to make processor 101 operate the software translator, and a program to make processor 101 operate the software interpreter), which procedure will be described afterwards. The apparatus of the present embodiment uses Java bytecode which is the instruction set of the Java virtual machine as the non-native code. The routine itself which implements the procedure of the method call of non-native codes is constituted by native codes.

Processing unit 106 has the function of the functional unit of M32R which is a 32-bit microprocessor made by Mitsubishi Electric Corporation. Instruction set of processing unit 106 are shown in Figs. 2-4. The details of the M32R processor are described in "Mitsubishi 32 Bit Single Chip Microcomputer M32R Family Software Manual (Rev. 1.1)". It will be understood that, although M32R is used in the present embodiment, the present invention is not limited thereto. One skilled in the art will understand that a data processor including a translator can easily be realized similarly from the following description even if a functional unit of

another processor is employed.

Referring to Fig. 5, bus 107 includes a data bus 107A and an address bus 107B. Multifunction instruction decoder 105 includes a hardware translator 120 connected to data bus 107A to convert non-native code applied from data bus 107A into a native code of processing unit 106, a multiplexer 121 having a first input receiving the output of hardware translator 120 and a second input connected to data bus 107A, which selects and provides to processing unit 106 the signal of one of the two inputs according to the value of a control signal, and a comparison circuit 122 connected to address bus 107B to compare the address on address bus 107B with an address that defines the storage region of non-native code and provide a control signal to control multiplexer 121 so that the output of hardware translator 120 is selected when the address on address bus 107B is an address of a storage region of non-native code, and the input from data bus 107A is selected otherwise.

Multifunction instruction decoder 105 converts the instruction, if it is non-native code, which is read out from the address (on address bus 107B) specified by the program counter into a native code of processing unit 106, and provides the result to processing unit 106. This process will be described in detail afterwards with reference to Fig. 10.

Fig. 6 shows Java bytecodes of a method to be converted in the description of a conversion example of the Java bytecode according to the present embodiment. According to Java, an intermediate code for the virtual machine which is the execution environment of Java is generated by compiling the source program. This intermediate code is executed in the virtual machine execution environment prepared for each real machine. This intermediate code is called "Java bytecode" since the length is variable in byte units.

Fig. 7 is a flow chart of the program execution procedure of the data processor according to the first embodiment. To execute a program described in the Java bytecode, the method of the Java bytecode that is to be initially executed after a pre-execution preparation process 301 is called and executed (302). Since a subroutine is called a "method" in the Java

bytecode, a subroutine in the present invention will be generally termed as "method" in the following description.

Fig. 8 is a flow chart of the procedure which calls a method of Java bytecode corresponding to step 302 of Fig. 7. It is assumed that, when the method of Java bytecodes is converted into a subroutine of native codes for execution by a software translator, the native code of the converted result is stored in the software cache set in RAM 103. In this case, by reading out the resultant native code of conversion from the software cache for execution when the same non-native code is invoked, the conversion process is bypassed to improve the execution speed.

Therefore, in the procedure which invokes a subroutine of non-native codes, detection is first made whether the converted result generated when this method was called previously remains in RAM 103 or not (401). If the converted result remains in RAM 103, that resultant method of conversion is invoked from RAM 103 to be executed (405).

If the converted result of this method does not remain in RAM 103, a process is carried out to select one of the hardware translator, software translator and software interpreter to be used to execute this method at step 402. The control branches at step 403 to the selected execution mechanism.

When execution by the hardware translator is selected, the subroutine of non-native codes is called (406). Then, the non-native code read out from the address of the program counter by multifunction instruction decoder 105 is converted into a native code, which is executed by processing unit 106 (Fig. 10).

When execution by the software interpreter is selected, the software interpreter subroutine is called with the read out code as an argument (407). The subroutine interprets the non-native code one code at a time for execution.

When execution by the software translator is selected, control proceeds to step 404 to convert the subroutine of the non-native code that is to be called into the subroutine of native codes. The converted result is first stored in RAM 103. Then, the process of invoking the subroutine of the converted native code by accessing RAM 103 is executed (405).

Fig. 9 is a flow chart of the procedure which selects the subroutine execution mode corresponding to step 402 of Fig. 8. Either execution with the hardware translator or execution with the software translator is selected depending upon the status of the software cache and the execution frequency of the subroutine that is to be executed. More specifically, when the invocation counts of the subroutine to be executed exceeds a predetermined number of times, execution with the software translator is selected, otherwise with the hardware translator is selected. In the present embodiment, this number of times is classified depending on whether there is an available region in the software cache in which the resultant native code of conversion is stored (this case the number is called "N") or not (this case the number is called "M"). Naturally, only an instruction that can be translated by the hardware translator is taken as the subject of translation by the hardware.

Referring to the routine of Fig. 9, determination is first made whether there is an empty region in the software cache (417). When there is an available region, control proceeds to step 418, otherwise to step 420.

At step 418, determination is made whether the number of invocation counts of the routine that is to be invoked is greater than the aforementioned N or not. When the number of invocation counts is greater than N, control proceeds to step 422, otherwise to step 419.

At step 419, determination is made to use the hardware translator. A flag for branching control not shown is set, and the process ends.

When there is no available region in the software cache, determination is made at step 420 whether the number of invocation counts of the subroutine that is to be invoked is larger than the aforementioned M or not. When the number of invocation counts is larger than M, control proceeds to step 422, otherwise to step 419.

In the case where there is an empty region in the software cache and the number of invocation counts of the subroutine is greater than N, or in the case where there is no available region in the software cache and the number of invocation counts of the subroutine is greater than M, control proceeds to step 422 to release a portion of the subroutine of the native code

in the software cache to make an empty region available in the software cache.

5 Then at step 423, determination is made whether an empty region for the processed result has been made available at step 422 or not. When an empty region has been made available, determination to use the software translator is made (421). When an empty region could not be made available, determination to use the software interpreter is made (424). In either case, a flag to branch the process not shown has an appropriate value set, and the process ends.

10 Fig. 10 is a flow chart of the process of multifunction instruction decoder 105. Referring to Figs. 5 and 10, determination is made through comparison circuit 122 whether the address range of the program counter output on address bus 107B is within the predetermined space where non-native codes are stored (432). If the result is YES, the code read out from  
15 the address specified by the program counter is regarded as non-native code to be converted into a native code by hardware translator 120 (434). The resultant native code is provided to processing unit 106 (435). In the example of Fig. 5, the code on data bus 107A is converted into a native code by hardware translator 120, and the result is selected by multiplexer 121 to  
20 be provided to processing unit 106.

When determination is made that the address range of the program counter are not within the space where the non-native code is stored at step 432, the code read out from the program counter is directly provided to processing unit 106 since it is a native code (433). In the example of Fig. 5,  
25 multiplexer 121 selects the code on data bus 107B, not the output of hardware translator 120, and provides the selected code to processing unit 106.

Conversion of non-native code into a native code at step 434 of Fig. 10 is schematically set forth below. In the case of a general non-native code,  
30 one or a plurality of native codes that are sequentially executed are set in correspondence to one non-native code in advance. Furthermore, the corresponding relationship between the operand of non-native code and the operand in a native code is preset. In the case where the non-native code is

for a virtual machine such as Java, the stack is to be assigned to the stack region of the memory or the register. Then, the value of the program counter is incremented with an appropriate number according to the number of native codes required to carry out a process corresponding to the execution of one instruction of non-native code. This is the basic conversion process.

Figs. 11 and 12 show a conversion process for non-native code that requires a complicated process (i.e., the size of the native code after conversion is large), and a conversion process for invokestatic, which is non-native code for method invocation, as exceptional examples. Refer to Fig. 6 as to non-native codes (Java bytecode) such as invokestatic.

Conversion is carried out according to the process as shown in Fig. 11 for a complicated non-native code (such as fadd). More specifically, a native code is generated that invokes a software routine (prestored in ROM 103) to execute the process of the non-native code that is to be converted (436). As to the non-native code of invokestatic, a processing routine thereof is similarly invoked as shown in Fig. 12. It is to be noted that this processing routine further carries out the procedure which calls a method of the non-native code shown in Fig. 8. This means that, during execution of the process of Fig. 12, the process of Fig. 8 is repeatedly executed, whereby the program of the non-native code is executed. Thus, the processing routine shown in Fig. 8 is executed every time a method is invoked in the present embodiment, including the case where an instruction that calls another method of non-native code is present in one method. Particularly, selection of the hardware translator, software translator and software interpreter is effected.

Similarly in the case of using the software translator, the native code generated by conversion corresponding to Java bytecode for method invocation will carry out the procedure which calls a method of the non-native code shown in Fig. 8. The program of the non-native code is executed by the repetitive execution of Fig. 8.

Similarly in the case of using the software interpreter, the procedure which calls a method of the non-native code shown in Fig. 8 is carried out

corresponding to the Java bytecode for method invocation. By executing the process of Fig. 8 repeatedly, the program of the non-native code is executed.

The procedure which calls a method of non-native code shown in Fig. 8 is prestored in ROM 103.

Fig. 13 is a flow chart of the procedure of a pre-execution process corresponding to step 301 of Fig. 7. The contents of the execution preparation process includes a step to allocate the stack region used by the program in RAM 102 (501), and the step of setting a stack pointer in a register (502). The value of the last address in the stack region plus 4 is set at the SP register of processing unit 106 in processor 10.

Fig. 14 is a flow chart of the procedure which converts a method of the Java bytecode into a subroutine of native codes (step 404 of Fig. 8). Referring to Fig. 14, a conversion starting step is effected (601), followed by the step of register allocation (602), and the step of method conversion (603). Each of these steps will be described hereinafter with reference to Figs. 15-20.

Fig. 15 is a flow chart of the procedure of a conversion starting step 601. At step 701, the head address of the Java bytecode which is to be converted, i.e., the start address of the method, is assigned to variable jpcstart, and the last address + 1 is assigned to variable jpcend.

Then, the memory for the local variable and the stack operand (702) is allocated. The number of the registers that can be allocated is assigned to set "REG" (703). The stack depth list is cleared (704). Variable RSi is set to 0 with respect to iteration control variable i (i=0 to nStack-1) (705). Similarly, variable RLj is set to 0 for iteration control variable j (j=0 to nLocal-1) (706). Here, nStack is the number of stack operands and nLocal is the number of local variables.

Fig. 16 is a flow chart of the procedure of register allocation step 602 of Fig. 14. In the process of Fig. 16, steps 804-812 are repeatedly processed for the Java bytecode in the method with address jpc altered from jpcstart to jpcend.

At step 801, the stack depth registration list is cleared. Since the

stack depth is 0 at  $\text{jpc}=\text{jpcstart}$ , i.e., at the start of the method execution, a set of the address  $\text{jpcstart}$  of the Java bytecode and its corresponding stack depth at that address, or  $[0, 0]$  in this case, is registered in the stack depth list (not shown). The contents of this process will be described afterwards.

5 At step 803,  $\text{jpc}$  is initialized to  $\text{jpcstart}$ .

Steps 804-813 are iterated with address  $\text{jpc}$  altered from  $\text{jpcstart}$  to  $\text{jpcend}$ .

10 At steps 804-806, the stack depth  $\text{js}$  at the execution of  $\text{jpc}$  (the position of the stack indicating the stack pointer when execution of the Java bytecode of address  $\text{jpc}$  ends) is computed. At step 804, determination is made whether address  $\text{jpc}$  and its corresponding stack depth at  $\text{jpc}$  is already registered. If there is not,  $\text{jsnext}$  is assigned to  $\text{js}$  at step 805, and control proceeds to step 807. If there is a registered stack depth, the stack depth registered in correspondence with  $\text{jpc}$  is assigned to  $\text{js}$  at step 806.

15 Then, control proceeds to 807.

At step 807, the next address  $\text{jpcnext}$  is computed. More specifically, at step 807, the Java bytecode at address  $\text{jpc}$  is assigned to  $\text{jinst}$ , the code size of  $\text{jinst}$  is assigned to variable  $\text{jinstsize}$ , and  $\text{jpc}+\text{jinstsize}$  is assigned to  $\text{jpcnext}$ . The  $\text{jinst}$  and  $\text{jinstsize}$  obtained at step 807 are the  
20 Java bytecode at  $\text{jpc}$  and the size (byte unit) of that Java bytecode, respectively.

When the stack depth at  $\text{jpc}$  is  $\text{js}$ , the stack depth after execution of  $\text{jinst}$  is computed by:

25  $\text{js} - \text{"the number of operands popped from the stack by jinst"} + \text{"the number of operands pushed on the stack by jinst"}$

Therefore, at step 808, the number of operands to be popped from the stack by  $\text{jinst}$  and the number of operands to be pushed on the stack by  $\text{jinst}$  are assigned to variable  $\text{consume}$  and variable  $\text{produce}$ , respectively. At step 809,  $\text{js}-\text{consume}+\text{produce}$  is assigned to variable  $\text{jsnext}$ .

30 At step 810, the computed  $\text{jsnext}$  is registered as the stack depth which corresponds to the address that has possibility of being executed succeeding  $\text{jinst}$  (the address of the next Java bytecode and address of branch target). At step 811, the reference count of the operand referred to

by jinst is recorded. At step 812, jpc is updated to the next address obtained. At step 813, determination is made whether  $jpc < jpcend$ . The process from step 804 to step 812 is repeated until jpc becomes jpcend.

5 Register allocation is effected at step 814, based on the reference count information obtained at step 811.

Fig. 17 is a flow chart of the procedure which registers the address and stack depth corresponding to steps 802 and 810 of Fig. 16. First, the address to be registered is assigned to variable jpcrecord, and the stack depth to be registered is assigned to variable jsrecord. Then, determination is made whether a stack depth corresponding to address jpcrecord is already registered (902). If a stack depth is not registered, the correspondence between address jpcrecord and stack depth jsrecord is registered (905). If a stack depth is already registered, the registered value is compared with the current value jsrecord to be recorded (903). If the two values match, the process ends. If the two values do not match, the stack operand cannot be allocated to the register since the stack depth differs depending upon the path to be executed (904). In this case, a flag not shown is set, which indicates register allocation is impossible for this method.

Fig. 18 is a flow chart of the procedure which records an operand reference count corresponding to step 811 of Fig. 16. First, the value of jsconsume is assigned to variable jsmin (1001). Reference count RSi is incremented by one for the stack operand to be popped (1002). Reference count RSi is incremented by one for the stack operand to be pushed (1003). Determination is made whether the local variable is to be referred to or not (1004). If the local variable is to be referred to (YES of 1004), reference count RLi of that local variable is incremented by one (1005).

Fig. 19 is a flow chart of the procedure of the register allocation determination process corresponding to step 814 of Fig. 16. First, determination is made whether the flag which may be set in step 904 in Fig. 17 indicates register allocation is impossible or not for this method. If register allocation is possible for this method, reference counts RSi and RLi are sorted (1104), registers are allocated in the descending order (1005). If register allocation is impossible for this method (YES at 1101), a register is

allocated similarly but only for the local variable (1102, 1103).

Fig. 20 is a flow chart of the procedure of a method conversion process corresponding to step 603 of Fig. 14. First, a native code is generated at the entry of the method. The details are shown in (1) of Fig. 37 (1201), and will be described afterwards. Then, the value of jpcstart is initialized with the value of variable jpc (1202).

The process of steps 1203-1207 is repeated sequentially for all the Java bytecodes in the method. More specifically, the Java bytecode at address jpc is assigned to jinst (1203). The code size of jinst is assigned to variable jinstsize (1204). Then, the address of the next code is computed by adding jpc and jinstsize and assigning the result to jpcnext (1205). Then, the stack depth, that is already registered corresponding to address jpc by the procedure of Fig. 16, is obtained and is assigned to variable js (1206). A native code is generated based on these obtained jpc, jinst and js (1207). At step 1208, jpcnext is assigned to variable jpc. At step 1209, the value of jpc is compared with the value of jpcend. If the value of jpc is smaller than jpcend, control proceeds to step 1203 to repeat the above-described process. If the value of jpcend is larger than jpc, the process ends.

Native codes as shown in Figs. 21-28 are generated according to the type of the Java bytecode at step 1207. In these figures and all figures thereafter, "TX" is the address of the native code generated with respect to the Java bytecode at address X. The native code generated changes depending upon whether the operands of each Java bytecode is allocated to the register or the memory.

"Epilogue code" in Fig. 27 will be described in detail in relation to the description of an actual conversion example. Also, the codes that push arguments shown in Fig. 28 will be described in detail in the section of the description of an actual conversion example. In Fig. 28, methodId is the address of the region where the character string representing the method to be called is stored. callJavaMethod is the routine that carries out the process of calling the method specified by the character string stored in the region specified by register R0 according to the procedure of Fig. 8.

Fig. 29 shows the usage of the register of the M32R in the subroutine

of a native code. Although the arguments of a subroutines are stored in a stack and delivered, some of the arguments can be stored in, for example, registers r0-r3 and delivered.

Figs. 30-33 show the usage of the stack in the subroutine of native codes. Fig. 30 represents the stack layout when a subroutine is invoked. Arguments are stored on the stack. Fig. 31 represents the stack layout after the first entry code in the subroutine is executed. From the status of Fig. 30, the region for local variables and the region for stack operands are reserved, and the values of the registers (r8-r14) that must be preserved before and after subroutine execution are pushed. In the figure, nStack is the number of stack operands, nLocal is the number of local variables, and nArg is the number of arguments. It is to be noted that the number of arguments is included in nLocal. For the example method shown in Fig. 6, nStack=6, nLocal=3, and nArg=2. The stack status is shown in Figs. 32 and 33.

The process of converting the method of the Java bytecode shown in Fig. 6 into the subroutine of native codes according to the processing procedure of Fig. 14 will be described hereinafter.

Following the execution of step 601, all operands are allocated to the memory region shown in Fig. 33. Fig. 34 represents the allocated status. For example, local variable [0] (first argument) is allocated to a memory region of 4 bytes that is offset from the stack pointer by 60 bytes.

At step 602, the process of Fig. 16 is executed. Figs. 35 and 36 show the way how the executing goes during the processes.

Immediately before the start of step 801, operand reference counts RS0-RS5 and RL0-RL2 are all set to 0. RS0-RS5 are the reference counts of stack operands [0]-[5]. RL0-RL2 are the reference counts of local variables [0]-[2]. At steps 801 and 802, [address 0, stack depth 0] is registered in the stack depth list. Although the stack depth list not shown, it can easily be implemented, for example, with an array of addresses indexed by the registered order, and an array of the stack depths indexed by the registered order. At step 803, jpc=0, jsnext=0 are set. This stage is shown in status (1).

10527 225600

Status (2) shows the stage when it comes to YES of step 813 after executing steps 804-813. It is noted that jpc succeeds the previous value 0 of status (1). Since [address 0, stacked depth 0] is registered in the stack depth list, steps 804 and 806 are executed, whereby js=0. The values of jinst, jinstsize and jpcnext computed at step 807 are iload\_0, 1 and 1, respectively. Since instruction iload\_0 pushes the value of local variable [0] on the stack, the values of consume and produce obtained at step 808 are 0 and 1, respectively. Therefore, jsnext computed at step 806 is js-consume+produce=0-0+1=1. At step 810, [address 1, stack depth 1] is registered since the Java bytecode of the next address jpcnext(=1) is the only Java bytecode that has the possibility of being executed subsequent to iload\_0. Since iload\_0 refers to local variable [0] (read) and stack operand [0] (write), corresponding reference counts RL0 and RS0 are respectively incremented by 1 to become 1 and 1, respectively, at step 811.

Similarly, by repeating steps 804-813 and arriving at the YES of step 813, the stages shown in status (3) (4) ... are achieved.

At status (8), [address 9, stack depth 1] and [address 21, stack depth 1] are registered in the stack depth list since the next execution of instruction ifge 21 may possibly be either address 9 which is the next address (jpcnext) or address 21 which is the branch target address.

At status (18), [address 28, stack depth 2] is registered since the next execution of instruction goto 28 corresponds to address 28 which is the branch target address.

At status (23), an attempt is made to register [address 28, stack depth 2] since the next execution of instruction invokestatic is the next address 28. However, since the stack depth for address 28 is already registered at status (18), the stack depth already registered and the stack depth to be registered are compared (step 902). Since the stack depths are same in this case, step 904 is skipped, and the registration process of the address and stack depth ends.

At status (25), nothing is registered in the stack depth list since the next possible address of instruction ireturn is determined only at the time of execution and not yet definite at this time point.

In this way, the reference counts of operands as shown in RS0-RS5 and RL0-RL2 of the last status (25) are obtained. Also, the stack depth indicated in the column of "instruction address and stack depth that are registered" of statuses (1)-(25) will be registered in the stack depth list.

5       At step 814, the process shown in Fig. 19 is carried out. From step 1101, control proceeds to step 1104. At step 1104, operand reference counts RS0-RS5 and RL0-RL2 obtained by that time are sorted in the descending order. In the present example, the order of RS1, RS0, RS2, RS3, RS4, RL0, RL2, RS5, RL1 is obtained. By allocating registers R8-R13 to the first six  
10       operands in this order at step 1105, the register allocation as shown in Fig. 34 is achieved. More specifically, register R8 is allocated to stack operand [1]. Register R9 is allocated to stack operand [0]. Register R10 is allocated to stack operand [2]. Register R11 is allocated to stack operand [3]. Register R12 is allocated to stack operand [4]. Register R13 is  
15       allocated to local variable [0].

Thus, step 602 ends, and control proceeds to step 603. At step 603, the process of Fig. 20 is implemented. Figs. 37 and 38 show native codes generated as the process is executed. In Figs. 37 and 38, the characters of S<0> to S<5>, L<0> to L<2> indicate the operands of Fig. 34. For example,  
20       "ldi S<0>, #1" corresponds to "ldi R9, #1", and "ld S<3>, @L<1>" corresponds to "ld R10, @(56, sp)".

At step 1201, the native code as shown in status (1) is generated. The contents of this native code correspond to the code which changes the stack layout from that of Fig. 32 to Fig. 33, and the code to load values from  
25       the memory into the registers for all the local variables that are arguments and one allocated to registers (here, L<0>). The meaning of nLocal, nStack and nArg are same as those employed in Figs. 30-33.

At step 1202, jpc=0. At step 1203, the Java bytecode jinst at the obtained address 0 is iload\_0. At step 1204, the computed jinstsize is 1.  
30       At step 1205, the computed jpcnext is 1. Since stack depth 0 at address 0 is registered as shown in (1) of Fig. 35, the obtained js is 0 at step 1202. At step 1207, native codes as shown in Figs. 21-28 are generated according to Java bytecode jinst that is to be converted. In the present example, the

native code as shown in Fig. 22 is generated since jinst is iload\_0. Since the operands at stack depth js=0 are S<0> and L<0>, and operands S<0> and L<0> are allocated registers R9 and R13, respectively, as shown in Fig. 34, the native code of mv S<0>, L<0> (i.e., mv R9, R13) indicated at the first line in Fig. 22 is generated. In this way, the native code shown at (2) in Fig. 37 is generated.

Thus, in a similar manner, the native codes of (3)-(25) in Figs. 37-38 are generated. Here, the epilogue codes of (25) which is generated for ireturn will bring back the stack layout from that of Fig. 33 to Fig. 32.

According to the program execution system of non-native code of the first embodiment of the present invention, by allocating registers to local variables and operand stacks, not only the number of resultant native codes after conversion is reduced but also the memory access frequency is reduced compared to the conventional case where memory is allocated. As a result, the execution speed is improved.

#### Second Embodiment

Many components of the structure of the data processor according to a second embodiment of the present invention are common to those of the first embodiment. According to the program execution system of the non-native code of the second embodiment, a register is allotted to the local variable and operand stack. The number of resultant native codes of conversion as well as the memory access frequency are reduced than in the case where the same are allotted a memory.

The contents from Figs. 1 to 18 are identical for both the first and second embodiments, except that the register determination process of step 814 in Fig. 12 is carried out according to the procedure of the flow chart of Fig. 39, not Fig. 19, in the apparatus of the second embodiment.

NS registers are allocated to the stack operands. In the present example, the four registers of R8-R11 are allocated, so that NS=4. For example, register R8 is allocated to stack operand [4n] (n=0, 1, ...). During execution, a native code is generated so that stack operands at from [b] to [js-1] are stored in the registers, where js is the stack depth. More specifically, register R8 holds stack operand [4n], where n is the maximum n

which satisfies  $4n < js$  during execution. Any stack operands allocated to register R8 but not held are temporarily stored in the memory. As for local variables, allocation similar to that of the first embodiment is implemented. More specifically, the reference counts of the local variables are sorted  
5 (2701), and registers are allocated in the descending order (2704).

Although the process flow itself of Fig. 20 is identical to that of the first embodiment, it is to be noted that the native code generation process for the Java bytecode of jinst at step 1207 is carried out according to the procedure of the flow charts of Figs. 40-42 depending upon the type of jinst,  
10 instead of the flow charts of Figs. 21-28.

Also, the usage of M32R register is as shown in Fig. 43 instead of Fig. 29.

Fig. 40 represents the native code generation process carried out for the Java bytecode that does not branch at step 1207 of Fig. 20. At step 2801,  
15 the stack operand to be used is obtained. The operands from [low] to [high-1] are used. At step 2802, the process of reserving the allocated registers for the operands from  $S < js >$  to  $S < high-1 >$  is carried out. Since operands from [b] to [js-1] are stored in the registers, in the case that generates registers other than these are to be used, the process of loading  
20 operands deeper than [b-1] from the memory to the register (2803, 2804) and the process of reserving the registers for operands shallower than [js] are carried out.

At step 2805, native codes shown in Figs. 21-28 (when stack operand corresponds to register) of the first embodiment are generated. The process  
25 of steps 2806, 2807 and 2809 makes the registers allocated for NS(4) stack operands at the stack top hold their values at the junction of the execution path (i.e. the address labeled as branching target). The register allocated to the stack operand does not always hold a correct value (although stack top to stack operand [b] are held in the registers, there is a possibility that the  
30 value of b differs for each execution path to the junction). If no measures are taken, the value in the register may be incorrect at the junction point of execution paths, so that the register can no longer be used at subsequent generation of the native codes. Therefore, native codes are generated at the

junction of execution paths so that topmost NS stack operands are always held in corresponding registers.

Fig. 41 is the native code generation process carried out for ifge at step 1207 of Fig. 20. Refer to (8) of Fig. 23 as for ifge. Since ifge refers to  $S<js-1>$ , this is loaded into the register (2902) if not yet held in the register (2901). Although the codes for conditional branching are to be eventually generated (2906), the native codes for junction must be inserted at 2904 and 2905. Since register  $S<js-1>$  may hold stack operand  $[js-5]$  by this native codes for junction,  $S<js-1>$  is moved into register r0 in advance (2903). At step 2904, the stack depth registered in correspondence with address jpcnext is assigned to variable jpnext. At step 2905, native codes are generated to make the NS stack operands from  $S<jsnext-1>$  below held in registers. At step 2906, a conditional branch instruction whose condition is based on register r0 is generated.

Fig. 42 is the native code generation process carried out for code goto at step 1207. Refer to (18) of Fig. 63 as for code goto. First, the native codes for junction is generated (3001, 3002). More specifically, in the present invention, js representing the stack depth of the branching destination is assigned to jsnext (3001). Then, the native codes to make the NS stack operands from  $S<jsnext-1>$  below held in registers are generated (3002).

Then, the branch instruction (here, "bra TX") is generated (3003). Since NS stack operands at the top of the stack are held in the registers at the next address, the value of b is changed (3004, 3005). More specifically, the stack depth registered corresponding to jpcnext is assigned to variable jsnext (3004). Then, the larger of the values of 0 and jsnext-NS is assigned to b (3005).

Fig. 44 shows the process of step 2802 of Fig. 40. In this process, a native code is generated to accommodate overflow of registers  $S<p>$  to  $S<q>$ . For this purpose, steps 3102-3105 is repeated for  $i=p$  to  $q$ .

First, the value of p is assigned to iteration control variable i (3101).

At the iteration portion, determination is made whether the value of stack operand  $[i-NS]$  is held in register  $S<i>$ . Since  $[b]$  to  $[js-1]$  are stored in

corresponding registers, the value of the stack operand will be retained if i-NS is between b and js-1. This determination is made at steps 3102 and 3103. At step 3102, determination is made whether i is equal to, or less than q. The process ends if i is greater than q. If i is equal to or less than q, control proceeds to steps 3103 to determine whether i-NS is less than b or not. If i-NS is less than b, control proceeds to step 3104, otherwise control proceeds to step 3102, skipping steps 3104 and 3105.

If i-NS is held in the register, the native code of "st S<i-NS>, @SAVE<i-NS>" is generated to store that value in the memory (3104). Then, b is modified to i-NS+1 in order to indicate that i-NS is no longer stored in the register (3105).

Fig. 45 shows the process of step 2804 of Fig. 40. Step 3204 is repeated for i=bottom to top. The process of steps 3201, 3203 and 3205 controls the iteration.

More specifically, determination is made whether bottom is equal to or less than top. If bottom is equal to or less than top, control proceeds to step 3202. If bottom is larger than top, the process ends.

At step 3202, the value of bottom is assigned to i. At step 3203, determination is made whether i is equal to or less than top. If i is equal to or less than top, control proceeds to step 3204, otherwise to step 3206. At step 3206, the value of bottom is assigned to b in order to indicate that the values up to bottom are held in the registers.

At step 3204, the native code of "ld S<i>, @SAVE<i>" to restore S<i> is generated. At step 3205, 1 is added to i, and control returns to step 3203. By the above-described process, native codes to accommodate underflow of registers S<bottom> to S<top> is generated.

The description follows which show how the method of Java bytecode of Fig. 6 is converted into a subroutine of native codes according to the procedure of Fig. 14 of the second embodiment.

The processes up to register determination process 814 are identical to those of the first embodiment. More specifically, following execution of step 601, all operands are allocated to the memory as shown in the column immediately preceding the conversion start process of Fig. 46. This is

similar to Fig. 34 of the first embodiment. At step 602, the process as shown in Fig. 35 is carried out according to the process of Fig. 16, as in the first embodiment. The reference counts of operands as indicated by RS0-RS5 and RL0-RL2 of the last status (25) are obtained, and the stack depth as indicated in the column of "instruction address and stack depth to be registered" of statuses (1)-(25) is registered.

At step 814, the process of Fig. 39 is implemented. At step 2701, control proceeds to step 2702 where the stack operands are allocated to four registers R8-R11. In the present example, stack operands [0] and [4] are allocated to register R8. Stack operands [1] and [5] are allocated to register R9. Stack operand [2] is allocated to register R1. Stack operand [3] is allocated to register R11. At step 2703, reference counts RL0-RL2 of the local variables are sorted in the descending order. In the present example, the order of RL0, RL2 and RL1 is achieved. At step 2704, the register allocation result as shown in Fig. 46 is obtained by allocating the first two local variables in this order to registers R12 and R13.

Thus, step 602 is completed, and control proceeds to step 603. The process of Fig. 20 is implemented at step 602. Figs. 47 and 48 show native codes generated during the execution of this process. In Fig. 47, the characters of S<0> to S<5>, L<0> to L<2> indicate the operands of Fig. 46. For example, ldi S<0>, #1 corresponds to ldi R8, #1 whereas ld S<3>, @L<1> corresponds to ld R10, @(56,sp).

At step 1201, the native code as shown in status (1) is generated. The contents of this code include the code to modify the stack layout from that of Fig. 32 to Fig. 33, and the code to load the value from the memory into the registers for any local variables that are arguments and that are allocated to registers (in the present example, L<0> and L<2>).

At step 1202, jpc=0. At step 1203, the Java bytecode jinst at address 0 obtained is iload\_0. At step 1204, the obtained jinstsize is 1. At step 1205, the obtained jpcnext is 1. Since stack depth 0 corresponding to address 0 is registered as shown in (1) of Fig. 35, the js obtained at step 1206 is 0. At step 1207, the process as shown in Figs. 40-42 is carried out corresponding to the Java bytecode jinst that is to be converted. In the

present example, the process of Fig. 40 is carried out since `jinst` is `iload_0`.

The values of `low` and `high` obtained at step 2801 of Fig. 40 are 0 and 1, respectively. At step 2802, the process of Fig. 44 is carried out. Here, control proceeds through steps 3101, 3102 (Yes), 3103, 3104 (No), 3107, 3102 (No) since `js=0`, `b=0`, `low=0`, `high=1`. Therefore, the process ends without generating a native code. Then, determination is made at step 2803, and control proceeds to the NO side. The process of step 2805 is similar to that of the first embodiment, and shown in Figs. 21-28. In the present example, native codes as shown in Fig. 22 are generated. Since the operands at stack depth `js=0` are `S<0>` and `L<0>`, and operand `S<0>` is allocated register R8 and operand `L<0>` is allocated register R12, the native code of "mv `S<0>`, `L<0>`" (i.e., mv R8, R12) shown at the first line in Fig. 22 is generated. At step 2806, determination is made whether there is a label in `ipcnext` (=1). Here, there is no label. Therefore, control proceeds to NO, and the process of Fig. 40 ends. In this way, the native codes at (2) in Fig. 47 are generated.

In a similar manner, the native codes of (3)-(7) of Fig. 47 are generated.

In the conversion process of (8) in Fig. 47, the process of Fig. 41 is carried out at step 1207. Since `js=2` and `b=0`, control proceeds to NO of step 2901. At step 2903, the native code of "mv `r0`, `S<1>`" is generated. Since stack depth 1 is registered for address 9 as shown in (8) of Fig. 35, the value of `jsnext` obtained at step 2904 is 1. At step 2905, the process of Fig. 49 is implemented. At step 3301, the process of Fig. 45 (`bottom=MAX(0, jsnext-NS)=0`, `top=b-1=-1`) is executed. Since the determination result at step 3201 is NO, the process ends. At step 2906, the native code of "bge `r0`, T21" is generated. In this way, the native codes at (8) are generated.

As to (9)-(11) of Fig. 47, the native codes shown in Fig. 47 are generated by a process similar to the process of (2).

In the conversion process of (12) in Fig. 47, the process of Fig. 40 is implemented at step 1207. At step 2801, the obtained values of `low` and `high` are 4 and 5, respectively. At step 2802, the process of Fig. 44 is carried out. Since `p=4`, `q=4`, `b=0`, `low=4`, `high=5` here, `i-NS=0` is achieved at step 3103, and the step of 3104 is executed. More specifically, a native code to

store stack operand [0] in the memory is generated. In the present example, the native code of "st S<0>, @SAVE<0>" is generated. Here, SAVE<0> represents the memory region allocated to S<0> at the beginning of the register allocation process. In the present example, it is (40, sp). After the native code is generated, control proceeds to step 3106, where b is updated to 1. At step 3107, determination at 3102 results in NO, and the process of Fig. 44 ends.

Returning to Fig. 40, control proceeds to the NO side from step 2803. Although the process of step 2805 is similar to that of the first embodiment, the native codes shown in Fig. 21 are generated since the current Java bytecode is iconst\_3. Since the operand at stack depth js=4 is S<4> here, and S<4> is allocated to register R8, a native code "ldi S<3>, #3" is generated, which is shown in Fig. 21, as the one for the case that the operand is allocated to a register. Thus, the process of generating the native codes of (12) in Fig. 47 is completed.

In the conversion process of (13) in Fig. 47, native codes are generated similar to (12). Here, the value of b is updated to 2 at step 3106.

At (14)-(16) in Fig. 47, native codes as shown are generated similar to (2).

In the conversion process of (17) of Fig. 47, the process of Fig. 40 is carried out at step 1207. The values of low and high obtained at step 2801 are 1 and 3, respectively. Although the process of Fig. 44 is implemented at step 2802, the determination result of step 3102 is NO, since p=3, q=2, low=1, high=3. Therefore, the process of Fig. 44 ends. At step 2803, control proceeds to step 2804. At step 2804, the process of Fig. 45 is implemented (top=b-1=1, bottom=low=1). From steps 3201, 3202 and 3203, i=1 is achieved, and step 3204 is executed. More specifically, a native code to reload stack operand [1] from the memory to the register is generated. In the present example, the native code of "ld S<1>, SAVE<1>" is generated. Here, SAVE<1> indicates the memory region (44, sp) allocated to S<1> at the beginning of register allocation.

Then, control proceeds through steps 3205 and 3203 (NO) to step 3206 where the value of b is updated to 1. The process of Fig. 45 ends.

Since the process of step 2805 is similar to that of the first embodiment, except that the current Java bytecode is imul, the native code of Fig. 24 is generated. Here, the operands at stack depth js=3 are S<2> and S<3>, that are allocated to registers R10 and R11, respectively. Therefore, the native code for registers allocated to both operands of Fig. 24, i.e. the native code of "mul S<2>, S<3>" is generated. Thus, the process of generating the native code of (17) of Fig. 47 is completed.

At the conversion process of (18) of Fig. 47, the process of Fig. 42 is carried out at step 1207. At step 3001, the obtained value of jsnext is 2. At step 3002, the process of Fig. 49 is carried out. At step 3301, the process of Fig. 45 is carried out (top=b-1=0, bottom=MAX(0, jsnext-NS)=0). From steps 3201, 3202 and 3203, i=0 is achieved, and step 3204 is executed. More specifically, the native code to reload stack operand [0] to the register from the memory is generated. In the present example, the native code of "ld S<0>, @SAVE<0>" is generated.

Control further proceeds through steps 3205 and 3203 (NO) to step 3206 where the value of b is modified to 0. The process of Fig. 45 ends. At step 3003, the native code of "bra T28" is generated. Since stack depth 1 is registered for address 21 as shown in (8) of Fig. 35, the obtained value of jsnext at step 3004 is 1. At step 3005, the value is modified as b=MAX(0, jsnext-NS)=0. The foregoing corresponds to the process of generating the native code of (18) in Fig. 47.

In a similar manner, the native codes shown in Figs. 47 and 48 are generated.

According to the program execution system of non-native code according to the second embodiment of the present invention, registers are allocated to the local variables and operand stacks. Accordingly, the number of resultant native codes of conversion as well as the memory access frequency are reduced in comparison to the case where memories are allocated to the local variables and operand stack. As a result, the execution rate is also improved.

#### Third Embodiment

In the data processor according to a third embodiment of the present

invention, a native code is not generated for the Java bytecode that only transfers data. The operand of a transfer destination is recorded in correspondence with the transfer source. When an operand that has the transfer source already recorded is to be used in the conversion of java  
5 bytecode that carries out an operation, a native code is generated that carried out an operation using the recorded transfer source. Therefore, the number of resultant native codes of conversion is reduced, whereby the execution speed is also improved.

The structure of the data processor according to the third  
10 embodiment has many elements common to those of the second embodiment.

As to Figs. 1-18 and 39, the process carried out by the data processor of the third embodiment is identical to the process carried out by the data processor of the second embodiment.

The usage of the M32R register is identical to that of Fig. 43.

15 Although the process flow of Fig. 20 per se is identical, the native code generation process for the Java bytecode of jinst at step 1207 is carried out by the procedure of the flow charts of Figs. 50-62 corresponding to each type of the Java bytecode instead of the flow chart of Figs. 40-42.

Fig. 50 represents the conversion process for Java bytecodes that  
20 only transfers data to a stack. These include, for example, the Java bytecodes of iconst\_<n> and iload\_<n>. Here, the location of the transfer source (or, the immediate value when the transfer source is an immediate value) is recorded into P<i> at step 3601 instead of generating a native code which actually executes the transfer. Here, P<i> (i=0 to nStack-1) is a data  
25 structure in which an immediate value or the location of a transfer source can be recorded. P<i> holds any of the values in the following Table 1. In the following description and drawings, the relationship between respective values and the meaning of P<i> are as follows.

Table 1

| Character           | Meaning  |
|---------------------|--|
| x (immediate value) | Indicates that the native code that loads immediate value x into stack operand [i] is pending.                   |
| L<n>                | Indicates that the native code that loads the value of local variable [n] into stack operand [i] is pending.     |
| S<i>                | Indicates that a valid value is held in the register allocated to stack operand [i].                             |
| SAVE<i>             | Indicates that the valid value held in stack operand [i] is stored in the memory, not in the allocated register. |
| -                   | Indicates that no recording of particular meaning is effected.   |

Step 3602 is the process to store the value in the registers allocated to the NS(4) entries at the stack top side of stack operands in the execution junction (i.e. the address labeled as branching destination). The register allocated to the stack operand does not always hold a correct value. If no measures are taken, the value in the register may be incorrect at the junction of execution paths, so that registers can no longer be used in the subsequent generation of native codes. To avoid this situation, native codes are generated at the junction of execution paths so that topmost NS stack operands are always held in corresponding registers. The native code for this purpose is inserted in the following cases.

When Java bytecode that causes execution to be succeeded to the next address is present immediately proceeding the label (i.e. when the label is present at the next address), the native code is inserted (by steps 3602, 3702, 3806 and 4003) immediately after (before the label) the native code with respect to that Java bytecode. The process of inserting this native code is as shown in Fig. 56.

In Java bytecode that effects branching, the native code is inserted immediately before generating the branch native code (by steps 4101-4102, 4203-4205, 4208-4210, and 4215-4217).

This native code insertion process is shown in Figs. 51 and 53.

Fig. 51 shows the process of generating a native code that actually effect transfer when the transfer of the immediate value or local variable to the stack operand is pending. By the control of steps 4401, 4402 and 4411, the process of steps 4403-4410 is repeated for i=0 to k. More specifically, the process to reserve register S<i> is carried out (steps 4404 and 4408) for

P<i> corresponding to an immediate value or L<n>. Then, the native code to load the immediate value or the value of L<n> to register S<i> is generated (4405, 4409). Then, the value of P<i> is modified to S<i> to indicate that a valid value is held in register S<i> (4406, 4410).

5       The process to generate native codes which reserves register S<k> (steps 3504 and 3704) is carried out according to the flow chart of Fig. 52. First, determination is made whether register S<k> is holding the value of stack operand [k-NS] (4602). If the value is not held, register S<k> is empty. If the value is held, the native code of "st S<k-NS>, @SAVE<k-NS>"  
10 is generated to save the value of register S<k> (identical to register S<k-NS>) (4603). Then, the value of P<k-NS> is modified to SAVE<k-NS> in order to indicate that value of register S<k-NS> is saved (4604).

15       Fig. 53 shows the process of generating native codes that load the operand that are saved in SAVE<i> and that are any of the NS operands at the stack bottom side from stack operand [k] into respective allocated registers. The process of steps 4503-4505 is repeated for i=k-NS (provided that 0 for k<NS) to k. More specifically, when P<i> is SAVE<i> (4504), the native code of "ld S<i>, @SAVE<i>" is generated (4504). The value of P<i> is modified to S<i> to indicate that a valid value is held in register S<i>  
20 (4505).

25       Fig. 54 is the generation process of a native code for iadd. Since iadd writes to writing into stack operand [js-2] (js is the stack depth before executing the iadd), the generation process of a native code to reserve the corresponding allocated register S<js-2> is carried out (3701). At step 3702, a native code is generated which adds the data of the transfer sources recorded in P<js-2> and P<js-1> and stores the added result into register S<js-2>. The particular native code generated here will be different as shown in Fig. 55 according to the combination of the values of P<js-2> and P<js-1>. There are special cases in which no native code is generated as in  
30 cases 1 and 6. In these two cases, the immediate value is recorded in P<js-2>. In other cases, P<js-2> is modified to S<js-2> in order to indicate that a valid value is stored in S<js-2>. The process following step 3703 is as shown in Fig. 56 (identical to step 3602 of Fig. 50).

Fig. 56 shows the native code insertion process for junction at jpcnext. When there is a label at jpcnext, i.e., when the program execution path is merged at jpcnext, the process of steps 4302-4304 is carried out.

At step 4302, the registered value of the stack depth at jpcnext is obtained and set to jpcnext. At step 4303, the native code generation process to commit the stack operands of [0] to [jsnext-1] is carried out (this process is as shown in Fig. 51). At step 4304, the native code generation process to load the NS stack operands from S<jsnext-1> into the registers is carried out. The process of this step is shown in Fig. 53.

A similar operation to iadd is carried out for isub, imul, and idiv except that the addition for iadd is replaced with subtraction, multiplication and division, respectively.

Fig. 57 is the native code generation process for invokestatic<int (int, int)>. Since invokestatic<int (int, int)> writes into stack operand [js-2] as iadd to store a return value, a native code generation process to reserve the allocated register S<js-2> as in iadd is carried out (3801). Then, the native code to push the data of the transfer source recorded in P<js-2> and P<js-1> on the stack as the argument is generated (3802, 3803). The particular native code generated will be different according to the values of P<js-2> and P<js-1>. The codes are not illustrated here but they are the same as iadd. The native code to be generated subsequently (3804, 3805) is similar to that of the first embodiment. The process of steps after step 3806 is shown in Fig. 56.

Fig. 58 is the native code generation process for ireturn. First, a native code to transfer the data of the transfer source recorded in P<js-1> to register r0 is generated (3901). The native code generated here varies according to the value of P<js-1>. Respective native codes thereof are not shown here. Native code 3902 that is generated subsequently is similar to that of the first embodiment. Finally, P<i> is set to be ready for the generation of a native code for the next addresses (3903, 3904).

Fig. 59 is the native code generation process for istore\_<n>. It is to be noted that istore\_<n> writes into local variable [n]. Prior to writing into the local variable, the data transfer to the operand stack that has this local

variable as the transfer source, when pending, must first be effected. For this purpose, a native code thereof is first generated (4001). Then, the native code to transfer the data of the transfer source recorded in  $P<js-1>$  to local variable  $[n]$  is generated. The native code generated here (not shown) varies depending whether the value of  $P<js-1>$  and whether  $L<n>$  corresponds to a memory or a register. The process of steps after step 4003 are shown in Fig. 56.

The process of step 4001 is shown in Fig. 60. By the control through steps 4801, 4802 and 4807, the process of steps 4803-4806 is repeated for  $i=0$  to  $js-1$ . More specifically, if  $L<n>$  is recorded in  $P<i>$  (4803), register  $S<i>$  is reserved (4804). The native code which loads  $L<n>$  to register  $S<i>$  is generated (4804).  $P<i>$  is modified to  $S<i>$  in order to indicate that a valid value is stored in  $S<i>$  (4806).

Fig. 61 is the native code generation process for goto. Prior to the generation of the branching native code (bra) (4103), the native code which stores the value into registers allocated to the NS operands at the stack top side from stack operand  $[js-1]$  is generated (4101, 4102). The process of these steps is shown in Fig. 52.

Fig. 62 shows the native code generation process for ifge. Before generating the native code which compares the data of the transfer source recorded in  $P<js-1>$  with 0, and the native code which conditionally branches if the transfer source data  $\geq 0$ , a native code which loads the value of the NS entries at the stack top into the allocated registers is generated. Here,  $k=js-2$  since the stack top is the stack top after the execution of ifge (steps 4203-4205, 4208-4210 and 4215-4217).

The description follows which shows how the method of Java bytecode of Fig. 6 is converted into a subroutine of native codes according to the procedure of Fig. 14.

The process up to step 602 is similar to that of the second embodiment. In other words, register allocation as shown in Fig. 46 is achieved.

At step 603, the process of Fig. 20 is carried out. Fig. 63 shows the native codes generated during the execution of this process.

At step 1201, the native code shown at status (1) in Fig. 63 is generated. This includes the native code to modify the stack layout from that of Fig. 32 to Fig. 33, and the native code to load any value that is an argument and allocated to the register (here, L<0> and L<2>) from the memory into the register.

At step 1202, jpc=0. At step 1203, the obtained Java bytecode jinst at address 0 is iload\_0. At step 1204, the obtained jinstsize is 1. At step 1205, the obtained jpcnext is 1. Since stack depth 0 is registered in correspondence to address 0 as shown in (1) of Fig. 35, the value of js obtained at step 1206 is 0.

At step 1207, a process shown in Figs. 50-62 is carried out according to Java bytecode jinst that is to be converted. In the present example, the process of Fig. 50 is carried out since jinst is iload\_0. At step 4001, L<0> is recorded into P<0>. Control proceeds to NO from step 3602, and the process of Fig. 50 ends. In this way, the status (2) of Fig. 63 is achieved. After all, no native code for iload\_0 is generated at (2).

Similarly for (3) of Fig. 63, no native code is generated. Only L<1> is recorded in P<1>.

In (4) of Fig. 63, the process of Fig. 54 is carried out at step 1207. At step 3701, the process of Fig. 52 is carried out (i=js-2=0). The determined result at step 4601 becomes "NO", and the process ends. At step 3702, the native code of Fig. 55 is generated. Here, the native code for case 29 is generated since operands P<0> and P<1> at stack depth js=2 are L<0> (register) and L<1> (register), respectively. The value of P<0> is altered to S<0>, and the process of Fig. 52 ends. At step 3703, the process of Fig. 56 is carried out. The determination result at step 4301 becomes "NO", and the process ends. Thus, the process of Fig. 54 ends.

In the conversion process of (5) in Fig. 63, the process of Fig. 59 is implemented at step 1207. At step 4001 of Fig. 59, the process of Fig. 60 is implemented. Since L<2> is not registered any of P<i> here, the determination result of step 4803 is always "NO". No native code is generated, and the process ends. Since P<0> is "S<0>" at stack depth js=1, a native code to store S<0> into local variable [2] at step 4002 is generated.

Since local variable [2] is allocated to the memory as shown in Fig. 46, the native code of "st S<0>, @L<2>" is generated. Although the process of Fig. 56 is implemented at step 4003, the determination result at step 4301 becomes NO, and the process ends. Thus, the process of Fig. 59 ends.

5       The conversion process for (6) of Fig. 63 is similar to (2). Immediate value 1 is recorded in P<0>, and no native code is generated.

Similarly for (7) of Fig. 63, L<0> is recorded in P<1>, and no native code is generated.

10       For (8) of Fig. 63, the process of Fig. 62 is implemented at step 1207. Since P<1> is L<0> at stack depth js=2 here, control proceeds to steps 4201, 4207 and 4208. At step 4208, the obtained jsnext is 1. At step 4209, the process of Fig. 51 is implemented (k=0). From steps 4401, and 4402, i=0 is achieved, and control proceeds to step 4403. Since P<0> is an immediate value, control proceeds to step 4404. At step 4404, the process of Fig. 52 is implemented (k=0). Control proceeds from step 4061 to the NO side, and the process of Fig. 52 ends. At step 4405, a native code is generated which loads the immediate value (1) of P<0> to S<0>. Here, the native code of "ldi S<0>, #1" is generated. At step 4406, P<0> is changed to S<0>. Control proceeds through steps 4406 and 4402 to NO, and the process of Fig. 51 ends.

15       ends.

20       ends.

At step 4205 of Fig. 62, the process of Fig. 53 is implemented (k=0). Since there is no P<i> that records SAVE<i>, the determination result at step 4503 become "NO". The process of Fig. 53 ends without the generation of a native code. At step 4206, the native code of "bra T21" is generated.

25       (9)-(13) of Fig. 63 are processed similar to (6) (2) (2) (6) (2), respectively. Immediate value 2, L<0>, L<1> and immediate value 3, L<2> are recorded in P<1> to P<5>, respectively.

30       In (14) of Fig. 63, the process of Fig. 54 is carried out at step 1207. Here, at step 3701 of Fig. 54, the process of Fig. 52 is implemented (k=js-2=4). Here, control proceeds through step 4601 (YES) and 4602. Since SAVE<0> is recorded in P<0>, control proceeds to step 4603. At step 4603, a native code which stores S<0> is generated. More specifically, the native code of "st S<0>, @SAVE<0>" is generated. At step 4604, the value of P<0>

is altered to the value of SAVE<0>, and the process of Fig. 52 ends.

At step 3702 of Fig. 54, the native code of Fig. 55 is generated. Since operands P<4> and P<5> correspond to immediate value 3 and L<2> (memory), respectively, at stack depth js=6 here, the native code of case number 5 is generated. The value of P<4> is altered to the value S<4>, and the process of Fig. 56 ends. Thus, the process of Fig. 54 ends.

In (15) of Fig. 63, the process of Fig. 54 is implemented at step 1207 (subtraction is used instead of addition). At step 3701, the process of Fig. 52 is carried out (k=js-2=3). Here, the determination result of step 4601 becomes "NO", and the process ends. At step 3702, a native code similar to that of Fig. 55 is generated. Since operands P<3> and P<4> are L<1> (register) and S<4> (register), respectively, at stack depth js=5 here, the native code of case number 27 is generated (except that div is used instead of add). The value of P<3> is altered to the value of S<3>.

Although the process of Fig. 56 is carried out at step 3703, the determination result of step 4301 becomes "NO", and the process ends. Thus, the process of Fig. 54 is completed.

In the conversion process of (16) of Fig. 63, the process of Fig. 54 is implemented at step 1207. Although the process of Fig. 52 (k=js-2=2) is carried out at step 3701 of Fig. 54, the determination result at step 4601 becomes "NO", and the process ends. At step 3702, the native code of Fig. 55 is generated. Since operands P<2> and P<3> are L<0> (register) and S<3> (register), respectively, at stack depth js=4 here, the native code of case number 27 is generated. The value of P<2> is changed to the value of S<2>. At step 3703, the process of Fig. 56 is implemented. The determination result at step 4301 becomes "NO", and the process ends. Thus, the process of Fig. 54 ends.

In the conversion process of (17) of Fig. 63, the process of Fig. 54 is implemented at step 1207 (except that multiplication is effected instead of addition). At step 3701, the process of Fig. 52 is carried out (k=js-2=1). At step 4601, the determination result becomes "NO", and the process of Fig. 52 ends.

Returning to Fig. 54, the native code of Fig. 55 is generated at step

3702 (except that mul is used instead of add). Since operands P<1> and P<2> are immediate value 2 and S<2> (register), respectively, at stack depth js=3, the native code of case number 2 is generated. Since the multiplication of immediate value 2 is obtained by shifting leftwards by two bits, instruction sll3, not instruction mul, is generated. The value of P<1> is altered to the value of S<1>. At step 3703, the process of Fig. 56 is implemented. At step 4301, the determination result becomes "NO", and the process of Fig. 56 ends with no operation. Thus, the process of Fig. 54 ends.

In the conversion process of (18) of Fig. 63, the process of Fig. 61 is implemented at step 1207. At step 4101 of Fig. 61, the process of Fig. 51 is effected ( $k=js-1=1$ ). Since P<i> recorded with an immediate value or a local variable is not present, control proceeds through steps 4403 (NO) and 4407 (NO) for both cases of  $i=0$  and 1. The process of Fig. 51 ends.

Returning to Fig. 61, the process of Fig. 53 is implemented at step 4102 ( $k=js-1=1$ ). At step 4501, the obtained  $i$  is 0. Control proceeds to steps 4502 and 4503. Since SAVE<0> is recorded in P<0>, control proceeds to step 4504. At step 4504, a native code is generated which load the value from the memory region of SAVE<0> into register S<0>. More specifically, the native code of ld S<0>, @SAVE<0> is generated. At step 4504, the value of P<0> is altered to S<0>. At step 4506,  $i=1$  is achieved, and control proceeds to steps 4502 and 4503. Since P<1> is S<1>, the determination result of step 4503 becomes "NO". Therefore, at step 4506,  $i=2$  is achieved. The determination result of step 4502 becomes "NO", and the process of Fig. 53 ends.

Returning to Fig. 61, the native code of "bra T28" is generated at step 4103. In this way, the status of (18) in Fig. 63 is achieved.

The value of jsnext obtained at step 4104 of Fig. 61 is 1. At step 4105, P<0> becomes S<0>. The foregoing corresponds to (18') in Fig. 63.

In the conversion process of (19) in Fig. 64, L<0> is recorded in P<1> similar to (2). No native code is generated.

At (20) of Fig. 64, 1 is recorded in P<2>, similar to (6), and no native code is generated.

At (21) of Fig. 64, the native code shown is generated similar to (4) (provided that subtraction is effected instead of addition). This corresponds to case number=25 of Fig. 55. P<1> is altered to S<1>.

At (22) of Fig. 64, L<2> is recorded in P<2>, similar to (2). No  
5 native code is generated.

At (23) of Fig. 63, the process of Fig. 57 is implemented at step 1207. At step 3801 of Fig. 57, the process of Fig. 52 is implemented (k=js-2=1). The determination result of step 4601 becomes "NO", and the process ends. Control returns to step 3802 of Fig. 57. Since P<1> is S<1>, the native code  
10 generated at step 3802 becomes "push S<1>". Since P<2> is L<2> and L<2> is allocated the memory, the native code generated at step 3803 of Fig. 57 is "ld r0, @L<2>" and "push r0". Control proceeds through steps 3804 and 3805, whereby the native codes shown is generated. Although the process of Fig. 56 is implemented at step 3806, no native code is generated.

In the conversion process of (24) of Fig. 64, the native codes shown  
15 are generated similar to (4). This corresponds to case number 14 of Fig. 55. P<0> is altered to S<0>.

In the conversion process of (25), the process of Fig. 58 is implemented at step 1207. Since P<0> is S<0>, the native code generated  
20 at step 3901 is "mv r0, S<0>". Control proceeds to step 3902, and the native code of (25) of Fig. 64 is generated.

According to the data processor of the third embodiment, no native code is generated for the Java bytecode that effects only data transfer. The transfer source is recorded in correspondence with the operand of the  
25 transfer destination. In the case where an operand that has the transfer source already recorded is to be used in the conversion of Java bytecode that effects an operation, a native code is generated that carries out the operation using the recorded transfer source. Therefore, the number of resultant native codes after conversion is reduced. As a result, the execution speed is  
30 also improved.

Although the present invention has been described and illustrated in detail, it is clearly understood that the same is by way of illustration and example only and is not to be taken by way of limitation; the spirit and scope

of the present invention being limited only by the terms of the appended claims.

0995837-13901  
106217-285660